Towards an Epistemology for Software Representations

Christopher A. Welty

Computer Science Dept.
Vassar College
Poughkeepsie, NY 12601
weltyc@cs.vassar.edu

Abstract

The KBSE community is actively engaged in finding ways to represent software and the activities that relate to various stages in its lifecycle. While the wealth of modeling activities have, necessarily, been founded on first order logic based representations, this paper reports on research into Software Information Systems that has found the domain of software knowledge to be inherently second order. A facility for accurately representing second order constructs such as are found in the software domain is also presented.

Keywords*: Knowledge Representation, Domain Modeling, Program Understanding, Software Information Systems.*

1 Introduction

The fields of Knowledge Representation, Domain Modeling, and KBSE deal primarily with representation systems that are first order. Often the users of these systems take for granted the fact that their representations are limited to first order, and forget that the world is full of knowledge requiring higher order reasoning.

This paper begins with a brief but motivated review of the nature of first order representations, and a few second order extensions that exist in certain representation systems. A representation problem that arose when studying ways to make Software Information Systems more effective is then presented, and a case is made that the source of the problem is the need for second order reasoning. Finally, a facility for supporting limited second order reasoning is described.

The main goal of this paper is not to propose a new representation system, but to make the point that software representations are inherently second order, and that regardless of the approach taken, this fact should be considered to insure the accuracy of a representation.

2 First Order Representations

Most symbolic representation systems are based on First Order Logic (FOL), and thus have two basic kinds of symbols: *predicate symbols* and *object symbols* [Carnap, 1961]. Object symbols denote instances or individuals, and predicate symbols denote properties or attributes of those individuals.

Although actual usage varies, the logical foundation is clear: *set membership is a unary predicate*, and therefore the name of a set is a predicate symbol [Carnap, 1947] [Quine, 1964]. This point may seem obvious or irrelevant, but the simple fact is that many practitioners ignore it, and it does come into play in the realm of software representations.

2.1 Classes, Instances, and Links

A common representation in FOL is something like the following:

Eagle(E1)
Number(10)
Age(E1,10)

Those accustomed to reading representations will interpret this as, "***E1*** is an ***Eagle***, ***10*** is a ***Number***, and ***E1***'s ***age*** is ***10***." This is not actually what it says according to the semantics of FOL, but because most representations follow this general scheme, implemented representation systems (such as are provided by object-oriented languages or frame-based languages) present scaled down first order systems which allow for the definition of three special kinds of symbols: *classes*, *instances*, and *links*.

An instance is similar to an object symbol in FOL except that it must be the member of some class, as with ***E1*** and ***10***.

A class is a special unary predicate that denotes a set, as with ***Eagle*** and ***Number***. A link is a binary predicate that represents a relationship between two instances, as with ***age***. The interpretation given above would be correct for a system with these constructs.

2.2 Superclass Inheritance

Another common representation in FOL is:

$\forall x$ ***Eagle(x)*** $\rightarrow$ ***Bird(x)***

This would be interpreted as "All eagles are birds," which again is not entirely correct according to the semantics of FOL, but is used so frequently to mean precisely this that representation languages almost universally supply a shorthand notation for expressing this taxonomic relationship, called *subclass*.

Although the subclass relationship is usually expressed as a relationship between two classes, e.g ***Eagle*** is a subclass of ***Bird***, it is worthwhile to note that, by definition, relationships between predicate symbols (classes are, as discussed above, predicate symbols) are *second order*, and it is important computationally to maintain the first order status of a representation system [Gödel, 1931]. The subclass relationship really is no more than a shorthand notation for the inference shown above, with a slightly different interpretation.

Clearly, via *modus ponens,* the result of making ***Eagle*** a subclass of ***Bird*** would be that the instance ***E1*** in the previous example would now be inferred to be an instance of ***Bird***. This is known as *superclass inheritance*.

2.3 Identifying Non-First Order Objects

The previous section mentioned the importance of maintaining a first order representation in a system. There are two common pitfalls of representing a domain that can cause a second order construct.

2.3.1 Instances of Instances. Most representation languages do not permit an instance to have instances. Consider the implications of such a construct in FOL:

Eagle(E1)
E1(E2)

When ***E1*** is used as a predicate symbol, the predicate ***Eagle*** becomes second-order because it is the predicate of a predicate. A common pitfall of modeling in FOL is to create a two-place predicate for instance, such as:

instance(E1, Eagle)
instance(E2, E1)

Which, syntactically, is first order. It is not, however, completely first order because, as stated in the beginning of this section, *set membership is a unary predicate*. This example violates the semantics of a first order system [Carnap, 1947] [Quine, 1964].

The point here is that, despite syntactic hacks like making a predicate called ***instance***, an instance of an instance is a second-order construct.

2.3.2 Links between Classes. First order representation languages also do not allow links between classes. A link is a two place predicate, and a class is a one place predicate, making the ***PreysOn*** link second order:

Pigeon(P1)
Eagle(E1)
PreysOn(Eagle, Pigeon)
PreysOn(Eagle, P1)

A class can not, therefore, be predicated by a link. It is common to speak of the relationship between an instance and its class as a link, but this must be understood to be different than a link between two instances.

2.4 Extensions to First Order Systems

Second order systems are generally avoided because they are undecidable. Many representation systems, however, provide small extensions that, while second order, are tightly controlled to avoid undecidability.

2.4.1 Smalltalk Meta-Classes. Smalltalk [Goldberg and Robson, 1983] provides the ability for classes to have certain properties of instances. They can, like instances, be sent messages and have their own variables, which are defined as part of the *meta-class* description of the class. While this is second order, this aspect of the representation is not subject to inference, (Smalltalk provides only for superclass inheritance) and undecidability is not a problem..

Smalltalk classes are also themselves instances of a special class named ***class***, making them instances which can have instances. Again, this second order relationship is controlled because there is no actual inference involved, it is provided more to keep the syntax cleaner, and make the message passing paradigm pervasive in the language.

2.4.2 Classic Meta-Individuals. Classic [Brachman, et al., 1991], a modern descendent of KL-ONE [Brachman and Schmolze, 1985], employs an approach to representing links on classes that has been called the *abstraction relationship* [Brachman, 1983].

This approach involves creating, as part of the language, a special kind of instance for each class which represents *the class as an object*. While these instances (called *meta-individuals*) behave like all other instances (having links and being instances of a concept called ***concept***), they have a special relationship, the abstraction relationship, between themselves and the class they represent. Each class in Classic, then, can have instances and a meta-individual.

Again, as with Smalltalk, this second order relationship does not introduce undecidability because it is not used in any inference.

3 Representing Software

This work is the result of studying *Software Information Systems* [Devanbu, Selfridge, and Brachman, 1990] in order to determine how to make them more effective [Welty, 1995]. A Software Information System (SIS) is a knowledge-based system which serves to make software maintenance less time-consuming by providing faster and more intelligent access to the software.

An SIS contains two representational parts: a code model and a domain model [Selfridge, 1990]. The former represents objects in the code (the software domain), which facilitates access to these objects, and the latter represents objects in the application domain, which facilitates understanding that domain.

3.1 Objects in the Application Domain

Knowledge of the application domain has long been recognized as a critical part of software maintenance [Curtis, Iscoe, and Krasner, 1988]. Representing some of this knowledge in a domain model is a fairly common practice, to assist in understanding during any phase of the software lifecycle [Iscoe, 1991].

Modeling a domain requires building an ontology for that domain, the specifics of which are dependant on the representation system being used. Typical elements of a domain ontology are classes, a class hierarchy, links, and rules which can infer these links between instances.

An example class hierarchy from the application domain of email distribution is shown in Figure 1. An instance of ***mail-message*** would be an electronic mail message ***from*** some instance of ***mail-sender to*** some instance of ***mail-recipient***, where ***from*** and ***to*** are links.

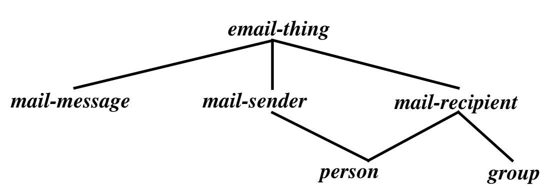

FIGURE 1. A simple domain hierarchy.

The goal of a domain model in SISs, and in software engineering in general, is to provide an accurate description of what the software "knows" about the objects in the domain *in a form which can be accessed by a maintainer.*

This latter point is critical in distinguishing domain-oriented techniques from the more generic software representations common to KBSE: while all software inaccessibly includes domain knowledge, a domain-oriented system treats the domain knowledge as distinct, and provides mechanisms for understanding it. Many KBSE systems take for granted the fact that a user will be a domain expert, but experience has shown this is frequently not the case [Curtis, Iscoe, and Krasner, 1988]. Providing the capability to understand the domain, is thus an important practical goal [Devanbu, Selfridge, and Brachman, 1990].

3.2 Objects in the Software Domain

The software domain contains objects like functions, data-types, and variables. It also contains assignment statements, for and while loops, if statements, parameters, etc. These are all the constructs defined by the programming language in which the software to be represented is written, and they are some of the classes in an ontology for code-level knowledge. Part of a possible class hierarchy for this domain is shown in Figure 2.

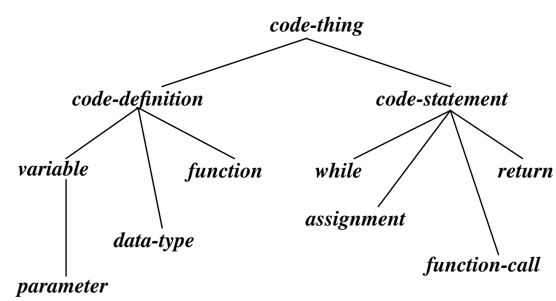

FIGURE 2. A taxonomy of code-level concepts.

Instances of these classes would be lines of code, variables, and the aggregation of variables and lines of code into functions, etc. For example, consider the following function in C:

```
void deliver_message_to_group (message,group)
MAIL_MESSAGE message;
GROUP group;
{ LIST members;

  members = get_members(group);
  while (! empty(members)) {
    deliver_message_to_person(message,
                              first(members));
    members = butfirst(members);
  }
}
```

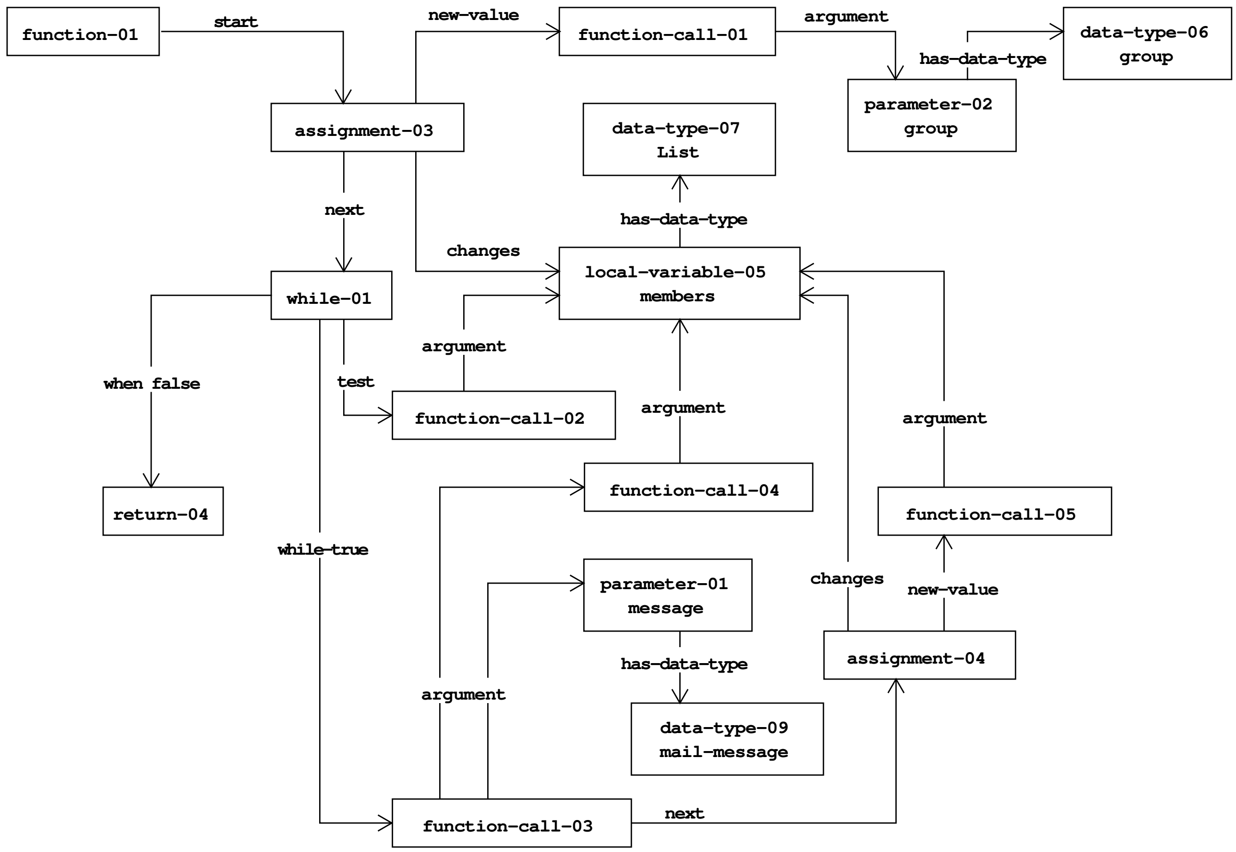

FIGURE 3. A semantic network view of a C function.

This entire function can be *completely* described as instances of the classes in Figure 2, since each of the C-language statements has a very rigid form that can be represented as links. For example, an assignment statement always has a variable which is changed (the left-hand side of the "="), and a new value which is either another variable or a function of another variable or variables (the right-hand side). An assignment statement, then, has two links: one which relates it to the variable to be changed, and one that relates it to another variable or to a function call. A semantic network view of the C function above represented in this way is shown in Figure 3, this is very similar to an *abstract syntax tree*.

Providing this level of representation for a program allows for significant benefits to maintainers engaged in understanding the program. For example, rules and other forms of inference have been employed to automatically detect side-effects, delocalized plans, vestigial code, and other common barriers to program understanding [Welty, 1995].

3.3 Integrating the Code and Domain Models

The LaSSIE SIS kept the code and domain models separate, and this led to two problems:

- There is implicit domain knowledge in the code model, and there is no way to verify that this knowledge is the same as what is explicitly represented in the domain model.
- When a maintainer has engaged in understanding a particular domain object through the domain model, the maintainer can not then move to the parts in the code model where that object is implemented.

For example, the objects ***mail-message*** and ***group*** appear both in the domain hierarchy shown in Figure 1 and the function representation shown in Figure 3. This is because conceptually they refer to the same thing. Groups and mail messages are objects in the domain that the program deals with directly.

3.3.1 Linking Object in Different Models. It is tempting to offer a solution to the above problems that simply "links" the objects in the domain model to the objects in the code model which implement them, i.e. somehow connect ***Group*** in the domain model to ***data-type-06*** in the code model.

There is a problem in linking these pairs. They clearly do represent the same concepts, however in the domain model they are classes and in the code model they are instances of the class ***data-type***, and a class can not be linked to an instance in a first order representation.

It may seem that making the domain model objects instances would solve this problem. This would allow the pairs to be linked, but what are the domain objects instances of, and what becomes of their instances? All the people, groups, and mail messages in the domain model would become instances of instances, which is also not allowed in a first order representation.

Numerous combinations and representation "hacks" can be (and have been) attempted to address this problem, but there actually is no first order solution. The reason is simply that *software representations are second order*. The correct representation is to make the domain objects instances of the class ***data-type***, and allow these instances to have instances of their own.

3.3.2 Existing Systems. At a glance, it would seem that this fits into the Smalltalk meta-class structure described in Section 2.4.1, but it does not for two reasons:

- Smalltalk does no inference with meta-classes. The whole purpose of representing the code-level knowledge this way was to employ inference to make information about the program more accessible to a maintainer trying to understand it.
- Data-types are not the only second order objects in the software domain. Functions, for example, can be represented in the domain model as plans [Devanbu and Litman, 1991].

Classic meta-individuals, described in Section 2.4.2, are also inadequate for this second order representation problem. Each class in the domain model could have a meta-individual which was linked to the corresponding instance of ***data-type*** in the code-model. This does provide part of the representation desired, but, again, there is no inference. There is a strong relationship between the domain model and code model objects, and one goal of integrating the two models is to verify that the corresponding objects accurately portray each other. When the program changes, the domain model must reflect that change.

4 Spanning Objects

A slightly more powerful second order extension to frame-based knowledge representation languages has been proposed [Welty and Ferrucci, 1994], and will eventually be available as an extension to Classic. This extension delves a little deeper into second order representation, allowing for some inferences, though still under tight control.

This extension, briefly, identifies first order predicates, that can themselves be predicated, as special objects called *spanning objects*. They are given this name because the representation is divided into two (or more) *universes of discourse*. One universe contains the spanning object as an instance, the other contains the spanning object as a class. The object spans these two universes through a *mapping function* that defines only the relationship between the two parts of the object. The mapping function can be set up e.g. to change the class in the one universe when the instance in the other universe changes.

Decidability problems are avoided with this approach by separating the objects that interact at the same level into distinct universes. No links are allowed between universes other than between the two parts of a spanning object via the mapping function.

4.1 Integrating Models Revisited

Spanning objects fully account for the problems of integrating the domain and code models. Instances of ***data-type*** and ***function*** (as well as a few others) are spanning objects which span the two universes representing the code and domain models, as shown in Figure 4. The mapping functions insure that the domain model classes accurately reflect their corresponding code model instances.

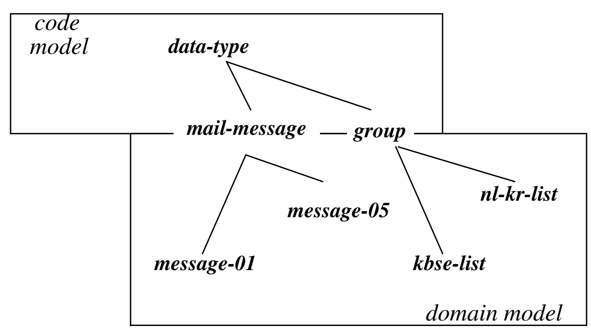

FIGURE 4. Spanning multiple universes.

In order for the mapping function to be able to generate accurate and useful domain objects, *all the domain knowledge must be represented in the code model*. In other words, the code model is a single model containing all the information needed to make the program run *and* all the information needed to help someone understand the domain.

4.1.1 Superclasses Aren't Enough. Considering the view in Figure 4, it may seem that a first order solution could be achieved by making ***group*** and ***mail-message*** subclasses of the class ***data-type***, rather than instances. There are several reasons why this will not work:

- Recall that in Section 2.3 it was stated that classes can not have links in a first order representation. Figure 3 shows the representation of a program as a set of instances, and the purpose of this representation was to allow inference that could assist a maintainer in understanding the program. If ***group*** and ***mail-message*** were classes rather than instances, the links shown and thus the inference would not be possible.

- ***Group*** and ***mail-message*** are simply not subclasses of ***data-type***. If there were, then by superclass inheritance their instances would also be instances of ***data-type***. Clearly a mail message or a group like those shown in Figure 4, are *not* data-types.

4.2 Epistemology

The second-order nature of the problem described here is a simple example of the *incompleteness* of formal languages: no language can represent itself.

The LaSSIE SIS had two models: a code model and a domain model, and each separate model was represented in Classic. The facilities of Classic were then used to make the information in these models more accessible.

The extended code-level ontology shown in Figure 2 is itself a programming language. Adding the domain knowledge to the code model in a way which supported the goals of a domain-centered approach (see Section 3.1) would have required additionally representing all the facilities of Classic *in the code-level ontology*. In other words, representing Classic in Classic.

5 Conclusion

Software understanding requires support not just for understanding the code, but the domain in which the software operates. The domain knowledge should therefore be represented explicitly and made available to maintainers as a distinct model.

In order to insure that this represented domain knowledge is consistent with the domain knowledge in the software, and to facilitate understanding where specific domain concepts appear in the software, it is desirable to include this domain knowledge as part of the software specification. Such a specification must use the domain knowledge in two ways: as part of the program and as a separate model for understanding the domain.

It was shown that supporting both these uses requires a second order representation, and a facility called spanning objects was presented that can represent integrated code and domain knowledge, and will be added to the next major release of Classic [Brachman, et al., 1991].

It is important to note that within the realm of KBSE systems, none has provided for both the ability to specify domain knowledge in a program *and* use that same specified knowledge to assist in understanding the domain. It is likely that one reason for this is that, as shown here, doing so requires a second order representation.

Acknowledgments

My thanks to Dave Ferrucci whose initial idea about multiple levels of instantiation (described in [Welty and Ferrucci, 1994]) led to the crystallization of these ideas regarding software. Thanks also to Peter Selfridge and Loren Terveen for comments and feedback.

References

[Brachman, 1983] Brachman, R. What IS-A is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer.* 16(10). Pp. 30-36. Oct, 1983.

[Brachman and Schmolze, 1985] Brachman, R. and Schmolze, J. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science.* 9(2). Pp. 171-216. 1985.

[Brachman, et al., 1991] Brachman, R., McGuinness, D., Patel-Schneider, P., Borgida, A. and Resnick, L. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. *Principles of Semantic Networks.* Morgan Kaufman. Pp. 401-456. May, 1991.

[Carnap, 1947] Carnap, R. *Meaning and Necessity.* U. of Chicago Press, 1947.

[Carnap, 1961] Carnap, R. *Introduction to Semantics and Formalization of Logic.* Harvard University Press, 1961.

[Curtis, Iscoe, and Krasner, 1988] Curtis, B., Iscoe, N. and Krasner, H. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM.* 31(11). Pp. 1268-1287. Nov, 1988.

[Devanbu, Selfridge, and Brachman, 1990] Devanbu, P., Selfridge, P., and Brachman, R. LaSSIE - A Classification-Based Software Information System. *Proceedings of the 12th International Conference on Software Engineering.* 1990.

[Devanbu and Litman, 1991] Devanbu, P. and Litman, D. Plan-Based Terminological Reasoning. *KR '91 Proceedings.* Morgan Kaufman. Pp. 128-137. 1991.

[Gödel, 1931] Gödel, K. Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme I. *Monatshefte für Mathematik und Physik*, Volume 38, pp 173-198. 1931.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley. 1983.

[Henninger, 1995] Henninger, S. An Organizational Learning Approach to Domain Analysis. *Proceedings of the 1995 International Conference on Software Engineering*. 1995.

[Iscoe, 1991] Iscoe, N., ed. *Proceedings of the ICSE 1991 Domain Modeling Workshop.* Austin Laboratory for Software Engineering and Computer Science. May, 1991.

[Quine, 1964] Quine, W. *From a logical point of view.* Harvard University Press, 1964.

[Selfridge and Brachman, 1990] Selfridge, P., and Brachman, R. Supporting a Knowledge-Based Software Information System with a Large Code Database. *Proceedings of the AAAI-90 Workshop on Knowledge-Base Management.* 1990.

[Selfridge, 1990] Selfridge, P. Integrating Code Knowledge with a Software Information System. *Proceedings of KBSA-5.* Pp. 183-195. Sept., 1990.

[Selfridge, 1991] Selfridge, P. Knowledge Representation Support for a Software Information System. *Proceedings of the Seventh Conference on Artificial Intelligence Applications.* Pp. 134-140. 1991.

[Welty and Ferrucci, 1994] Welty, C., and Ferrucci, D. *What's in an Instance?* RPI Computer Science Technical Report. 1994.

[Welty, 1995] Welty, C. *An Integrated Representation for Software Development and Discovery.* Ph.D. Thesis, RPI Computer Science Dept., Troy, NY. June, 1995.